Clase 6 — 16.12.25

#JAVA

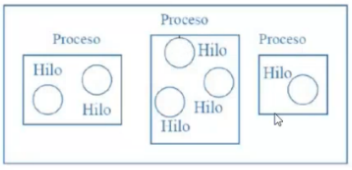
👨‍🏫 **Profesor:** José Antonio Martín
📘 **Unidad:** Programación Multiproceso
🗓️ **Fecha:** 16/12/2025
🎯 **Tema:** Programación Multihilo

🧵 Programación Multihilo

1️⃣ Introducción a los hilos (Threads)

🔹 ¿Qué es un hilo?

Un **hilo (thread)** es la **unidad mínima de ejecución** que puede ser planificada por el sistema operativo dentro de un proceso.
Mientras que el **proceso** representa un programa en ejecución con sus recursos asignados, el **hilo representa una línea concreta de ejecución** dentro de ese programa.



📌 Este esquema ilustra visualmente que:

- Un **proceso puede contener múltiples hilos**
- Los hilos **no existen fuera de un proceso**
- Varios procesos pueden ejecutarse a la vez, pero **no comparten hilos**

📌 La idea clave que transmite el dibujo es:

> **El hilo es una unidad interna al proceso, no una entidad independiente del sistema operativo.**
> Un mismo proceso puede contener **uno o varios hilos**, lo que permite que el programa realice **varias tareas de forma concurrente**, sin necesidad de crear nuevos procesos independientes.

Los hilos de un mismo proceso **comparten**:

- 🧠 **Espacio de direcciones (memoria)**
  Todos los hilos pueden acceder a las mismas variables, objetos y estructuras de datos del proceso.
- 📁 **Recursos del proceso**
  Incluye descriptores de archivos, sockets, recursos del sistema, etc.
- 📂 **Ficheros abiertos**
  Un archivo abierto por un hilo puede ser usado por otro hilo del mismo proceso.

📌 Esta compartición es **la gran ventaja**, pero también **la principal fuente de problemas** (condiciones de carrera, incoherencias, etc.).

📌 A pesar de compartir recursos, **cada hilo mantiene su propio contexto de ejecución**, lo que permite que se ejecuten de forma independiente:

- 📍 **Contador de programa (Program Counter)**
  Indica la instrucción exacta que está ejecutando el hilo en cada momento.
- 🧱 **Registros**
  Cada hilo tiene sus propios registros de CPU, evitando interferencias directas con otros hilos.
- 🧵 **Pila (stack)**
  Contiene las llamadas a métodos, variables locales y contexto de ejecución del hilo.

📌 Gracias a esta separación, varios hilos pueden ejecutar **diferentes métodos o diferentes partes del mismo método** simultáneamente.

🔹 Hilos vs Procesos

Procesos	Hilos
Independientes entre sí	Comparten memoria y recursos
Comunicación vía sistema operativo (IPC)	Comunicación directa en memoria
Cambio de contexto costoso	Cambio de contexto ligero
Mayor aislamiento	Menor aislamiento
Más seguros	Más eficientes

📌 **Idea clave**:

- Los **procesos priorizan seguridad y aislamiento**
- Los **hilos priorizan rendimiento y eficiencia**

📌 **Conclusión clave**:
Si una aplicación está formada por **tareas relacionadas que cooperan entre sí**, es **mucho más eficiente usar hilos** que múltiples procesos separados.

2️⃣ Multihilo vs Multiproceso

🔹 Multiproceso

El **multiproceso** hace referencia a la ejecución de **varios procesos independientes** de forma concurrente bajo el control del sistema operativo.

Características principales:

- 🧩 Cada proceso es un programa distinto
- 🔒 No comparten memoria por defecto
- 🔄 Se comunican mediante mecanismos del SO (pipes, sockets, señales, etc.)
- 🧠 El sistema operativo gestiona su planificación

Ejemplo típico:

- 🌐 Navegador web
- 🎵 Reproductor de música
- 📝 Editor de texto

Todos se ejecutan “a la vez”, pero **no tienen relación directa entre ellos**.

🔹 Multihilo

El **multihilo** se da cuando **un único proceso** contiene **varios hilos de ejecución** que cooperan entre sí.

Características principales:

- 🧠 Comparten memoria y recursos
- ⚡ Comunicación rápida entre tareas
- 🧩 Cada hilo suele tener una responsabilidad concreta
- ⚠️ Requiere control de concurrencia

Ejemplo típico dentro de una aplicación:

- 🧑‍💻 Un hilo gestiona la interfaz gráfica
- ⚙️ Otro realiza cálculos o procesamiento
- 💾 Otro guarda datos en segundo plano

📌 Para el usuario, el programa parece **más rápido y fluido**, aunque internamente esté repartiendo el trabajo entre hilos.

⚠️ **Pregunta típica de examen**:

> **Multiproceso ≠ Multihilo**

📌 No confundir:

- Multiproceso → varios programas
- Multihilo → un programa con varias tareas

3️⃣ Ventajas del uso de hilos

El uso de hilos aporta ventajas claras en **rendimiento, eficiencia y diseño de software**, especialmente en sistemas modernos.

Principales beneficios:

- 🕑 **Creación rápida**
  Crear un hilo dentro de un proceso existente es **mucho más rápido** que crear un proceso completo (aprox. 10 veces menos coste).
- 🧹 **Finalización ligera**
  Al terminar un hilo solo se libera su contexto y su pila, mientras que al terminar un proceso hay que eliminar todo su PCB y recursos asociados.
- 🔄 **Cambio de contexto eficiente**
  Cambiar de un hilo a otro del mismo proceso requiere menos operaciones que cambiar entre procesos distintos.
- 🚀 **Mejor aprovechamiento de la CPU**
  Permite que la CPU siga trabajando mientras un hilo espera por E/S o recursos.
- 🧠 **Diseño más modular**
  Cada hilo puede encargarse de una tarea concreta, mejorando la claridad y mantenibilidad del programa.

📌 En muchos sistemas operativos (Windows NT, Linux, OS/2) se resume así:

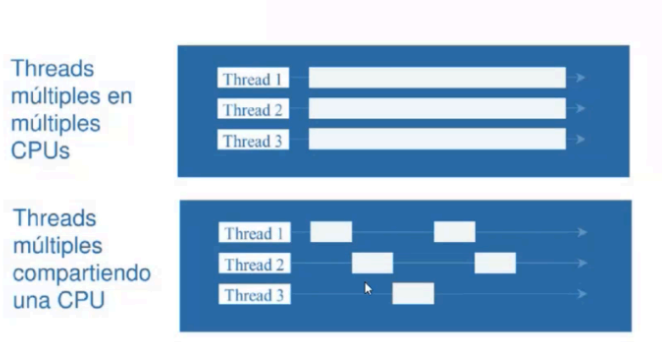
> *“Hilos baratos, procesos caros”*

4️⃣ Ejecución en CPU

🔹 Múltiples CPUs o núcleos

En sistemas con varios procesadores o núcleos:

- 🧵 Cada hilo puede ejecutarse **simultáneamente**
- Existe **paralelismo real**
- El rendimiento puede escalar con el número de núcleos



🔎 Qué representa el esquema

El esquema compara dos situaciones:

1. **Arriba**:
   - Cada hilo ocupa su propia CPU
   - Ejecución continua
   - Paralelismo real
2. **Abajo**:
   - Todos los hilos comparten una CPU
   - Ejecución fragmentada
   - Intercalado por el scheduler

🧠 Comentario detallado para los apuntes

📌 El esquema aclara una confusión muy común:

- Multihilo **NO implica siempre paralelismo real**
- Con una sola CPU hay **concurrencia**, no paralelismo

📌 En una sola CPU:

- El sistema **pausa y reanuda hilos**
- El usuario percibe simultaneidad
- La CPU solo ejecuta **un hilo cada instante**

📌 En múltiples CPUs:

- Los hilos pueden ejecutarse **literalmente al mismo tiempo**

📌 Ideal para:

- Cálculo intensivo
- Servidores
- Aplicaciones concurrentes reales

🔹 Una sola CPU

En sistemas con una única CPU:

- Los hilos **no se ejecutan al mismo tiempo**
- Se **intercalan** mediante el *scheduler*
- Se crea una ilusión de simultaneidad

📌 El sistema operativo decide:

- Qué hilo se ejecuta
- Cuánto tiempo
- Cuándo se interrumpe

📌 Importante recordar:

- ❌ El orden de ejecución **NO está garantizado**
- 🎲 Depende del planificador del sistema operativo
- 🔁 Puede variar en cada ejecución del programa

5️⃣ Aplicaciones multihilo

El uso de hilos es especialmente útil cuando una aplicación debe **responder al usuario** mientras realiza otras tareas.

🔹 Casos de uso habituales

- 🧑‍💻 **Interfaz + procesamiento en segundo plano**
  Evita que la aplicación se “congele”.
- 🔄 **Procesamiento asíncrono**
  Tareas que no deben bloquear el flujo principal.
- ⚡ **Aceleración de tareas**
  Dividir trabajo en partes ejecutables en paralelo.
- 🧩 **Estructuración modular del programa**
  Cada hilo representa una responsabilidad clara.

📌 **Ejemplo típico de aplicación multihilo**:

- 🎨 Un hilo dibuja menús e interfaz
- ⚙️ Otro procesa datos
- 💾 Otro guarda automáticamente información

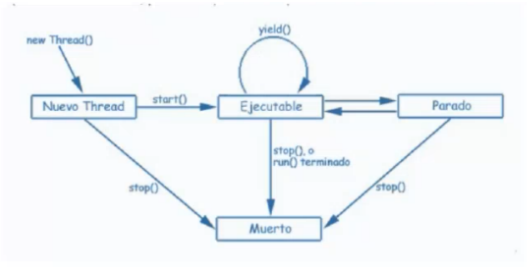
👉 Para el usuario: programa fluido
👉 Internamente: tareas repartidas entre hilos

6️⃣ Estados de un hilo (visión general)

🔹 Estados principales

El **estado de un hilo** describe **en qué situación se encuentra respecto al uso de la CPU** y a su ciclo de vida dentro del sistema.

A alto nivel, un hilo puede estar en uno de estos **cuatro grandes estados conceptuales**:



🔎 Qué representa el esquema

Es un **diagrama de estados** del hilo con:

- Nuevo
- Ejecutable
- Parado
- Muerto

Y las transiciones mediante:

- `start()`
- `yield()`
- `stop()`
- finalización de `run()`

🧠 Comentario detallado para los apuntes

📌 El esquema muestra que un hilo:

- **No se ejecuta al crearse** (estado *Nuevo*)
- Solo pasa a ejecutable cuando se llama a `start()`
- Puede entrar y salir del estado *Ejecutable* varias veces
- **Muere definitivamente** al terminar `run()`

📌 Detalle importante:

- Un hilo **no vuelve nunca** del estado *Muerto*
- El estado *Parado* representa una **interrupción temporal**, no una finalización

⚠️ Idea de examen:

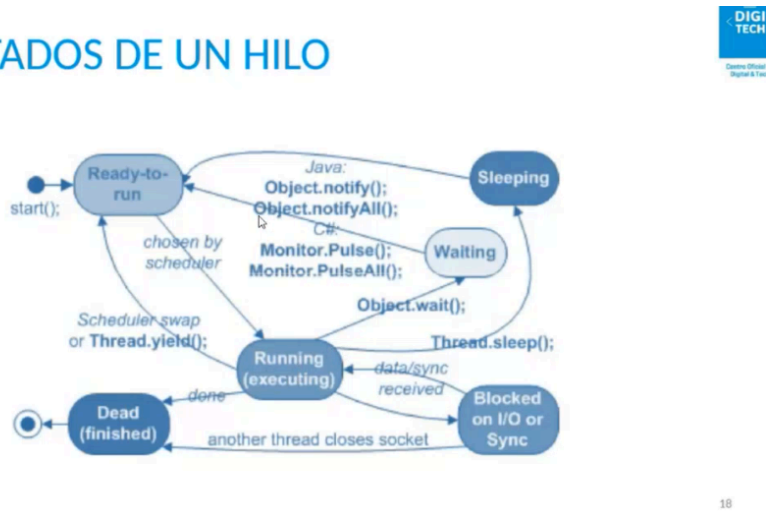
> Un hilo puede pasar varias veces por *Ejecutable*, pero solo una vez por *Muerto*.

📌 Esta clasificación es **didáctica**: internamente, los sistemas operativos manejan más estados, pero esta abstracción es clave para **entender y razonar** sobre el comportamiento de los hilos.

📌 Idea importante:

> Un hilo puede estar **vivo sin estar ejecutándose**.

7️⃣ Estados de un hilo (detalle)



📌 El esquema muestra que:

- Un hilo **no ejecuta continuamente**
- Puede detenerse por:
  - tiempo (`sleep`)
  - espera activa (`wait`)
  - bloqueo de recursos (E/S)

📌 Transiciones importantes:

- `yield()` → cede la CPU voluntariamente
- `notify()` → despierta hilos en espera
- Al finalizar `run()` → pasa a *Dead*

📌 Aclaración clave:

> Un hilo puede estar **vivo pero no ejecutándose**.

🔹 Nuevo

Un hilo se encuentra en estado **Nuevo** cuando:

- Ha sido **creado en memoria**
- Existe el objeto `Thread`
- **Aún no ha comenzado su ejecución**

Ejemplo conceptual:

```
Thread h = new Thread(r);
```

📌 En este estado:

- El hilo **no compite por la CPU**
- El sistema operativo **no lo planifica**
- Solo existe como estructura de datos

⚠️ Punto clave de examen:

> Crear un hilo **NO implica** que empiece a ejecutarse.

🔹 Ejecutable (Vivo)

Este estado indica que el hilo está **activo** y forma parte del conjunto de hilos gestionados por el scheduler.

Incluye **dos situaciones distintas**:

🟢 Preparado (Ready-to-run)

- El hilo **está listo para ejecutarse**
- Tiene todo lo necesario
- Está **esperando turno de CPU**

📌 Puede permanecer aquí:

- Porque otros hilos tienen prioridad
- Porque la CPU está ocupada

🔵 Ejecutándose (Running)

- El hilo **tiene el control de la CPU**
- Está ejecutando instrucciones en ese instante
- Solo **un hilo por núcleo** puede estar en este estado

📌 El cambio entre *Preparado* y *Ejecutándose* lo decide exclusivamente el **scheduler** del sistema operativo.

⚠️ Importante:

> El programador **no controla** directamente cuándo un hilo pasa a ejecutarse.

🔹 No ejecutable

Un hilo entra en estado **No ejecutable** cuando **no puede usar la CPU temporalmente**, aunque siga estando vivo.

Este estado **no es definitivo**: el hilo puede volver a Ejecutable.

😴 Dormido — `sleep(t)`

- El hilo se suspende **durante un tiempo fijo**
- No consume CPU
- Al finalizar el tiempo → vuelve a *Preparado*

📌 Uso típico:

- Pausas controladas
- Simulación de tiempos
- Esperas temporales

⏸ Esperando — `wait()`

- El hilo queda **a la espera de una señal**
- No se reanuda por tiempo
- Necesita:
  - `notify()`
  - o `notifyAll()`

📌 Uso típico:

- Coordinación entre hilos
- Comunicación productor–consumidor

🔒 Bloqueado — E/S o sincronización

- El hilo espera:
  - Entrada/salida (disco, red…)
  - Liberación de un recurso
  - Acceso a una sección crítica

📌 Mientras está bloqueado:

- No puede continuar
- El sistema puede ejecutar otros hilos

📌 Idea clave:

> El estado *No ejecutable* permite que la CPU **no quede desperdiciada** mientras un hilo espera.

🔹 Muerto / Finalizado

Un hilo entra en estado **Muerto** cuando:

- El método `run()` finaliza
- No hay más instrucciones que ejecutar
- El hilo libera:
  - Su pila
  - Su contexto de ejecución

📌 En este estado:

- El hilo **ya no existe como hilo activo**
- **No puede reiniciarse**
- Llamar otra vez a `start()` provoca error

⚠️ Punto de examen:

> Un hilo solo puede morir **una vez** y **no puede revivir**.

8️⃣ Transiciones importantes

Las **transiciones** representan los cambios de estado provocados por métodos o eventos.

- `start()`
  👉 **Nuevo → Ejecutable**
  Activa el hilo y lo registra en el scheduler.
- `yield()`
  👉 **Ejecutándose → Preparado**
  El hilo cede voluntariamente la CPU.
- `sleep()`
  👉 **Ejecutándose → Dormido**
  Suspensión temporal controlada.
- `wait()`
  👉 **Ejecutándose → Esperando**
  Suspensión hasta recibir notificación.
- `notify()` / `notifyAll()`
  👉 **Esperando → Preparado**
  Reactiva uno o varios hilos en espera.
- Fin de `run()`
  👉 **Muerto**
  Final definitivo del hilo.

📌 Importante:

> No existe garantía de cuándo volverá a ejecutarse un hilo despertado.

9️⃣ Consultar el estado de un hilo

Java permite **consultar el estado** de un hilo en tiempo de ejecución:

```
hilo.getState();
hilo.isAlive();
```

🔹 `getState()`

Devuelve un valor del enum `Thread.State`, útil para:

- Depuración
- Monitorización
- Diagnóstico

🔹 `isAlive()`

Indica si el hilo está **activo en el sistema**.

📌 Devuelve:

- `true`
  → Ejecutable o No ejecutable
- `false`
  → Nuevo o Muerto

📌 Muy importante:

> `isAlive()` **NO significa** “se está ejecutando ahora”.

🔟 Creación y puesta en marcha de hilos en Java

Java proporciona distintas herramientas para trabajar con hilos, organizadas en paquetes.

🔹 Paquetes implicados

`java.lang`

Paquete básico, siempre disponible.

Incluye:

- `Thread`
  Representa el hilo como entidad de ejecución.
- `Runnable`
  Representa la tarea que ejecuta el hilo.
- `ThreadGroup`
  Permite agrupar hilos para gestión conjunta.
- `ThreadDeath`
  Relacionada con errores en hilos (uso poco común).

📌 Este paquete cubre lo **esencial y clásico** del multihilo.

`java.util.concurrent`

Paquete avanzado para concurrencia moderna.

Incluye:

- Mecanismos de sincronización
- Colas seguras
- Variables atómicas
- Locks avanzados

📌 Diseñado para:

- Evitar errores clásicos
- Simplificar código concurrente
- Mejorar escalabilidad

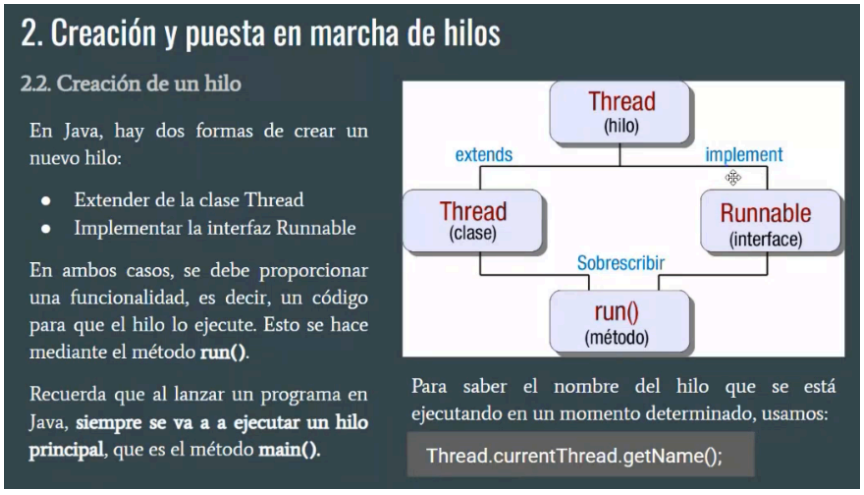
1️⃣1️⃣ Formas de crear un hilo en Java

Opción 1️⃣ Extender `Thread`

En este enfoque, la clase **es el hilo**.

```
class Hilo1 extends Thread {
    public void run() {
        // Código del hilo
    }
}
```

```
Hilo1 h = new Hilo1();
h.start();
```



📌 Características:

- Sencillo de entender
- Menos flexible
- Bloquea la herencia

⚠️ Por eso se considera **menos recomendable**.

Opción 2️⃣ Implementar `Runnable` ✅ (RECOMENDADA)

Aquí se separa claramente:

- 🧠 **La tarea** (`Runnable`)
- 🧵 **El hilo** (`Thread`)

```
class Ejec1 implements Runnable {
    public void run() {
        // Código del hilo
    }
}
```

```
Ejec1 r = new Ejec1();
Thread h = new Thread(r);
h.start();
```

📌 Ventajas clave:

- ✅ No bloquea la herencia
- 🔁 Permite reutilizar la tarea
- 🧩 Separación clara de responsabilidades
- 📚 Más alineado con `java.util.concurrent`

1️⃣2️⃣ El método `run()` vs `start()`

En Java, la diferencia entre `run()` y `start()` **no es sintáctica, es conceptual**. Ambos métodos existen en la clase `Thread` , pero **cumplen funciones completamente distintas** dentro del modelo de

concurrencia.

Cuando se llama al método `run()`, Java **no crea ningún hilo nuevo**. El método se ejecuta exactamente igual que cualquier otro método: en el **hilo que realiza la llamada**. Por defecto, esto suele ser el **hilo principal (`main`)**, por lo que no hay ejecución concurrente ni paralela. El sistema operativo **no interviene**, el scheduler **no participa** y el programa se comporta como si fuera monohilo.

```
hilo.run(); // ❌
```

📌 Esto provoca uno de los errores conceptuales más frecuentes:

> Pensar que `run()` "lanza" un hilo, cuando en realidad **solo ejecuta código**.

En cambio, cuando se llama a `start()`, se produce una secuencia completamente diferente. Java solicita al sistema operativo la **creación de un nuevo hilo de ejecución**, registra ese hilo en el **scheduler**, le asigna una pila propia y lo integra en el ciclo de planificación. Solo entonces, **el propio sistema** invoca internamente al método `run()`.

```
hilo.start(); // ✅
```

📌 El punto clave es este:

> `start()` **no ejecuta directamente** `run()`, sino que **crea un nuevo hilo que ejecutará `run()` cuando el scheduler lo decida**.

Por eso:

- `start()` → concurrencia real
- `run()` → ejecución secuencial

1️⃣3️⃣ Otras consideraciones importantes

🔹 Un hilo no puede reiniciarse

El ciclo de vida de un hilo está **estrictamente definido**. Un hilo se crea, se ejecuta y termina. Una vez que el método `run()` finaliza, el hilo pasa al estado **Muerto** y **no puede volver a utilizarse**.

Esto no es una limitación arbitraria, sino una decisión de diseño:

- El hilo ya ha liberado su pila
- Su contexto de ejecución ha desaparecido
- El sistema operativo lo considera terminado

📌 Por tanto:

> Si se quiere repetir una tarea, hay que **crear un hilo nuevo**, no reutilizar uno antiguo.

🔹 `start()` solo se puede llamar una vez

Este punto está directamente relacionado con el anterior. Llamar a `start()` implica:

- Crear el hilo a nivel de sistema
- Registrar el hilo en el scheduler

Una segunda llamada a `start()` sobre el mismo objeto `Thread` **rompe el modelo de ejecución**, por lo que Java lo prohíbe.

📌 De nuevo, es una regla de ciclo de vida, no de sintaxis.

🔹 El orden de ejecución no es predecible

En programación multihilo, el programador **no controla el orden exacto** en el que los hilos se ejecutan. Aunque el código esté escrito en un orden concreto, el sistema operativo puede:

- Interrumpir un hilo en cualquier momento
- Cambiar a otro hilo
- Reanudar el primero más tarde

Este comportamiento depende de:

- El scheduler
- La carga del sistema
- El número de núcleos
- Las prioridades de los hilos

📌 Consecuencia directa:

> El mismo programa puede producir resultados distintos en ejecuciones diferentes si no se controla la concurrencia.

🔹 Cambiar el nombre de un hilo

Aunque cambiar el nombre de un hilo no afecta a su ejecución, **sí afecta enormemente a la comprensión del programa**. En aplicaciones reales con decenas de hilos, identificar qué hilo está ejecutándose es fundamental.

Asignar nombres claros permite:

- Leer logs correctamente
- Depurar errores de concurrencia
- Seguir el flujo de ejecución

1️⃣4️⃣ Detener temporalmente un hilo

Un hilo **no está siempre ejecutándose**, y de hecho, la mayor parte del tiempo puede estar esperando. Esto es **normal y deseable**, ya que evita desperdiciar CPU.

Cuando un hilo entra en estado **No ejecutable**, no desaparece ni termina: simplemente **cede el uso de la CPU** hasta que se cumpla una condición concreta.

😴 `sleep(t)`

`sleep()` provoca una suspensión **voluntaria y temporal** del hilo. Durante ese tiempo:

- El hilo no ejecuta instrucciones
- No consume CPU
- No responde a eventos

Al terminar el tiempo indicado, el hilo **no vuelve directamente a ejecutarse**, sino que pasa al estado *Preparado*, donde espera turno de CPU.

📌 Importante:

> `sleep()` **no libera monitores ni recursos sincronizados**.

⏸ `wait()`

`wait()` es un mecanismo de **coordinación entre hilos**. Cuando un hilo ejecuta `wait()`:

- Se suspende indefinidamente
- Libera el monitor asociado
- Espera una notificación externa

Solo puede reanudarse mediante:

- `notify()`
- `notifyAll()`

📌 Esto permite que los hilos se comuniquen sin consumir CPU innecesariamente.

🔒 Bloqueo por E/S o sincronización

Un hilo puede quedar bloqueado cuando:

- Espera datos de entrada/salida
- Intenta acceder a un recurso protegido
- No puede entrar en una sección crítica

En este caso, el bloqueo **no lo decide el hilo**, sino el entorno en el que se ejecuta.

📦 1️⃣5️⃣ Paquete `java.util.concurrent`

Este paquete surge para **resolver los problemas clásicos del multihilo tradicional**: código difícil de leer, errores sutiles y sincronización manual compleja.

Su filosofía es:

> Proporcionar herramientas de alto nivel que gestionen la concurrencia **de forma segura y eficiente**.

🔹 Clases de sincronización

Estas clases permiten coordinar hilos sin necesidad de gestionar estados manualmente. Internamente, utilizan mecanismos avanzados del sistema, pero exponen una interfaz clara.

Por ejemplo:

- Un `CountDownLatch` permite que un hilo espere a que otros terminen
- Un `CyclicBarrier` obliga a varios hilos a sincronizarse en un punto común

📌 El programador expresa **la intención**, no la mecánica.

🔹 Colas concurrentes

Las colas bloqueantes permiten desacoplar hilos productores y consumidores. Un hilo puede insertar datos mientras otro los procesa, sin preocuparse de:

- Bloqueos manuales
- Estados inconsistentes
- Accesos simultáneos peligrosos

Esto mejora:

- Rendimiento
- Legibilidad
- Escalabilidad

🔹 Tipos atómicos

Los tipos atómicos permiten realizar operaciones complejas como incrementos **de forma indivisible**. Esto evita problemas clásicos como:

- Lecturas inconsistentes
- Condiciones de carrera

Todo ello sin bloquear otros hilos, lo que los hace muy eficientes.

🔹 Locks avanzados

Los locks modernos ofrecen un control más fino que `synchronized`, permitiendo, por ejemplo:

- Diferenciar lectura y escritura
- Evitar bloqueos innecesarios
- Mejorar el rendimiento en sistemas concurrentes

🧠 1️⃣7️⃣ Métodos para identificar el hilo en ejecución

En un programa multihilo, varias partes del código pueden ejecutarse **casi a la vez**, y el orden puede cambiar entre ejecuciones. Por eso, una de las primeras necesidades reales cuando trabajas con hilos es poder responder a preguntas como:

- ¿Qué hilo está ejecutando esta línea ahora mismo?
- ¿Este log lo escribió el hilo principal o un hilo worker?
- ¿Por qué una tarea se ejecuta dos veces o se “pisa” con otra?

Para eso sirve `Thread.currentThread()`: devuelve una referencia al **hilo que está ejecutando el código en ese instante**. No es “el hilo que creaste”, ni “el último hilo arrancado”, sino **el que actualmente está corriendo esa instrucción**.

```
Thread.currentThread();
```

📌 Conceptualmente, es como decir:

> “Dame el *yo* del hilo actual”.

Esto es muy útil porque en un entorno concurrente el mismo método `run()` (o cualquier método invocado desde él) puede ser ejecutado por distintos hilos. Si no identificas qué hilo está "dentro" en cada momento, el comportamiento parece aleatorio.

🔹 `getName()` — Identificación legible (humana)

```
Thread.currentThread().getName();
```

El nombre del hilo está orientado a **lectura humana**. Es especialmente útil para:

- 🗒️ **Logs**: saber qué hilo escribió cada mensaje
- 🧪 **Depuración**: entender secuencias de ejecución concurrente
- 🧩 **Trazabilidad**: cuando hay muchos hilos (pool, workers, etc.)

📌 En la práctica, un log sin nombre de hilo es como una novela sin narrador: se entiende… pero a base de sufrir.

Ejemplo conceptual de log:

- `[main] iniciando app`
- `[Worker-1] procesando tarea`
- `[Worker-2] esperando datos`

Esto permite detectar cosas como:

- Un hilo que se queda bloqueado siempre
- Un hilo que ejecuta tareas que no le corresponden
- Intercalados inesperados entre hilos

🔹 `getId()` — Identificación única (técnica)

```
Thread.currentThread().getId();
```

El `id` del hilo es un identificador numérico que sirve para:

- 🧠 **Diferenciar hilos aunque tengan nombres iguales**
- 🔍 **Diagnóstico** cuando se comparan trazas y volcados
- 📊 **Análisis** de comportamiento (por ejemplo, "el hilo 17 siempre se bloquea")

📌 Ventaja clave del `id`:

> Es estable como "identificador" durante la vida del hilo.

A diferencia del nombre, que puede ser poco informativo o repetirse si el programador no lo gestiona bien, el `id` sirve como etiqueta única.

✅ Cómo se usan juntos (idea de examen)

Lo habitual (y lo más útil) es combinar **nombre + id**, porque:

- Nombre → te orienta rápidamente
- Id → te asegura unicidad

📌 En resumen:

- `Thread.currentThread()` → “quién soy ahora”
- `.getName()` → “cómo me llamo”
- `.getId()` → “qué identificador único tengo”

🔄 1️⃣8️⃣ Transiciones avanzadas entre estados

Un hilo **no avanza como una flecha recta** desde “inicio” hasta “fin”. En realidad, su ejecución se parece más a un “rebote” constante entre estados, porque un hilo puede:

- Ejecutarse un rato
- Ceder la CPU
- Dormirse
- Esperar una señal
- Bloquearse por un recurso
- Volver a estar preparado
- Y repetir esto varias veces antes de terminar

Este comportamiento es justo lo que reflejan las transiciones avanzadas: muestran que el hilo es una entidad **dinámica**, gestionada por el scheduler y por condiciones del entorno (tiempo, recursos, sincronización).

🔹 `yield()` — Ceder CPU voluntariamente

`yield()` indica algo como:

> “Yo puedo seguir, pero si hay otros listos, que pasen primero”.

- Pasa de **Running** → **Ready**
- No bloquea, no duerme, no espera
- Solo cede turno

📌 Importante:

- No garantiza nada: el scheduler puede incluso volver a escogerlo de inmediato.

🔹 `sleep(t)` — Pausa temporal

`sleep(t)` provoca que el hilo se vuelva **no ejecutable** durante un tiempo.

- Pasa a **Sleeping**
- No consume CPU
- Al acabar el tiempo vuelve a **Ready**, no directamente a Running

📌 Detalle conceptual:

> Dormirse no significa “paro el programa”, significa “dejo de competir por CPU temporalmente”.

🔹 `wait()` + `notify()` / `notifyAll()` — Coordinación entre hilos

`wait()` es una espera **condicionada**, no por tiempo, sino por evento.

- `wait()` → el hilo entra en **Waiting**
- `notify()` → despierta **uno**
- `notifyAll()` → despierta **todos** los que esperen

📌 Matiz esencial:

- Despertar no implica ejecutar: al recibir `notify`, el hilo vuelve a **Ready** y espera CPU.

Esto explica un punto típico de confusión:

> "He hecho notify, ¿por qué no se ejecuta ya?"
> Porque notify solo lo devuelve a la cola de listos.

🔹 Bloqueo por E/S o sincronización

Hay transiciones que no dependen del hilo, sino del entorno:

- E/S: disco, red, teclado…
- Sincronización: espera a que se libere un monitor / recurso

En ese caso el hilo entra en **Blocked**, que conceptualmente significa:

> "Podría ejecutarse… si tuviera el recurso".

📌 Conclusión clave:

> La programación multihilo no puede basarse en suposiciones de orden.

Porque el orden depende de:

- Scheduler
- Recursos
- Señales (`notify`)
- Tiempo (`sleep`)
- Carga del sistema

🧠 1️⃣9️⃣ Aclaración clave sobre "Vivo"

El término "vivo" se usa para indicar si el hilo **sigue existiendo como entidad activa** dentro del sistema, no si está "trabajando" en ese momento.

Un hilo **vivo** puede estar:

- **Ejecutándose (Running)** → usando CPU ahora mismo
- **Preparado (Ready)** → listo pero esperando turno
- **Esperando (Waiting)** → esperando señal (`notify`)
- **Dormido (Sleeping)** → esperando tiempo (`sleep`)
- **Bloqueado (Blocked)** → esperando recurso (E/S o sincronización)

📌 Por tanto:

- **Vivo** = "no ha terminado"
- **Ejecutándose** = "está usando CPU"

Esto es crucial para entender por qué:

- `isAlive()` puede devolver `true` aunque el hilo parezca “parado”
- Un hilo puede estar vivo “sin hacer nada” porque está esperando o bloqueado

⚠️ Error típico:

> Confundir “vivo” con “ejecutándose”.

2️⃣0️⃣ Detener temporalmente un hilo (clasificación exacta)

Esta clasificación es importante porque pone orden a un concepto que suele mezclarse: “el hilo no corre”. En realidad, “no corre” puede significar cosas **muy distintas**, y cada una tiene implicaciones diferentes.

✅ 1) 😴 Dormido → `sleep(t)`

- Pausa por tiempo fijo
- No depende de otros hilos
- Cuando acaba el tiempo, vuelve a Ready

📌 Ideal para:

- pausas controladas
- temporización simple

✅ 2) ⏸ Esperando → `wait()` + `notify()/notifyAll()`

- Pausa por condición / evento
- Depende de otro hilo para reanudarse
- Se usa para coordinación

📌 Ideal para:

- comunicación entre hilos
- productor-consumidor clásico

✅ 3) 🔒 Bloqueado → E/S o sincronización

- Pausa porque falta un recurso
- Puede depender del hardware (E/S) o del propio programa (bloqueos)
- Es el tipo más “traicionero” porque a veces no es obvio dónde se queda bloqueado

📌 Ejemplos típicos:

- esperando datos de red
- esperando acceso a una sección crítica

📌 Idea clave final:

> “No ejecutable” no significa “apagado”, significa “no puede competir por CPU ahora”.